Reading types

```
data Bool = False | True
```

This type has two possible values. The pipe between them indicates exclusive disjunction: if you have a `Bool` value, it will either be `True` or `False`, never both.

```
data Maybe a = Nothing | Just a
```

This type takes another type as an argument; the `a` type argument is polymorphic and could be any type. `Maybe` also encodes the possibility of not returning a meaningful value, *xor* an `a` value wrapped inside a constructor.

```
data Either a b = Left a | Right b
```

Another disjunctive type, this time taking two polymorphic type arguments. It's important to note here that `a` and `b` *may* be different types but are *not required* to be.

```
data (,) a b = (,) a b
```

This type is not disjunctive; it is conjunctive. It requires both an `a` and a `b` argument in order to construct a value of this type. Again, `a` and `b` may be different types but are not required to be.

Function application with structure

```
-- Functor's fmap
(<$>) :: (a -> b) -> Maybe a -> Maybe b
```

Lifts the function `(a -> b)` into the `Maybe` structure, applies it to the `a` value inside, gives you a `Maybe b`. Of course, if the `Maybe` value was `Nothing`, you get `Nothing` out.

```
-- Applicative's tie-fighter
(<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
```

Accepts two arguments: a function which might exist and a value which might exist, if you're lucky. Applies the function to the second argument if *both* exist. Any `Nothing` means it's all `Nothing`.

```
-- Monad's bind
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

Similar to the `Applicative` except now the argument function might be producing *more* structure. If nothing is `Nothing`, then you'd end up with a `Just (Just a)` result. The magic of Monad is that `join` can smush that nested structure.

```
traverse :: (a -> IO b) -> [a] -> IO [b]
```

`IO` is the datatype we use when we'll be performing effects. You might find that you have a list of `IO` actions that, when performed, get you one response, but you really wanted one big `IO` action that would give you a list of responses. `traverse` to the rescue! `traverse` is `fmap` and `sequence` combined, where `sequence` flips `[IO b]` into `IO [b]`.

Comparing types

We can see patterns in common functions by matching up type signatures.

Function application

Apply a function, lifting over structure where necessary.

```
 ($)  ::                    (a ->  b)  ->  a  ->  b
(<$>) ::     Functor f =>   (a ->  b)  -> f a -> f b
(<*>) :: Applicative f => f (a ->  b)  -> f a -> f b
(=<<) ::       Monad f =>   (a -> f b) -> f a -> f b

mapM ::     (Monad f, Traversable t) =>
            (a -> f b) -> t a -> f (t b)
traverse :: (Applicative f, Traversable t) =>
            (a -> f b) -> t a -> f (t b)
```

Manipulating structure

Structural manipulation without applying or lifting a function first.

```
sequence :: (Monad f, Traversable t) =>
                   t (f a) -> f (t a)
join :: Monad f => f (f a) -> f    a
```

Bind and traverse are made of smaller parts

```
bind v f = join (fmap f v)
bind   :: Monad m => m a -> (a -> m b) -> m b
(>>=) :: Monad m => m a -> (a -> m b) -> m b

trav f v = sequenceA (fmap f v)
trav     :: (Applicative f, Traversable t) =>
            (a -> f b) -> t a -> f (t b)
traverse :: (Applicative f, Traversable t) =>
            (a -> f b) -> t a -> f (t b)
```

Composition

Function composition alongside Kleisli composition, that is, composition in the presence of additional structure.

```
 (.)  ::            (b -> c)   -> (a -> b)   -> a -> c
(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> a -> m c
```

(psst, composition is not as weird as it looks!)

```
(f . g) x = f (g x)


-- in GHCi
> ((+3) . (*10)) 4
> 43


-- lining up the arguments with the parameters
(.) :: (b -> c) -> (a -> b) -> a -> c
--       (+3)        (*10)     4    43
```